By Piet de Roo

How Do You Slice Your Pizza?

On the Unification of the Terminology and the Procedure for Equivalence Partitioning

Equivalence partitioning is one of the software test design techniques that really makes sense. So much sense that almost every tester applies this technique, some without even realizing that what they call "common sense" is actually a formal technique. Yet somehow we, as a testing community, do not seem to agree on what the procedure is for equivalence partitioning. We do not even agree on the terminology. Is a partition the same thing as a class? What does it mean that a class is valid or invalid? Once we know what the classes are, how do we combine them into test cases? Do we need output partitioning? In this article I will make a proposal for a unification of the terminology for equivalence partitioning and I will attempt to find a single method for deriving test cases. I gathered the knowledge of many test gurus and I looked for the common denominator, trying to remove the ambiguity without creating a mathematical challenge.

Sources

Quite a few sources were consulted to gather information on the EP technique. Most of the authors refer to Glenford Myers [1] and Boris Beizer [2]. Not all sources describe the full process and not all sources describe the same process. The most practical information on how to apply the technique has been described by Erik van Veenendaal [6] and Edward Kit [3].

Terminology

Bits and pieces

In equivalence partitioning we take the inputs to a computer program and chop them up into bits and pieces that are supposed to be handled in the same manner by the program.

Various sources use various terminology to describe these bits and pieces:

- Myers [1]: "The equivalence classes are identified by taking each input condition ... and partitioning it into two or more *groups*."
- Black [9]: "... equivalence *classes*, which are also called equivalence *partitions* ..."
- Van Veenendaal [6]: "... equivalence *classes* or *partitions* ..."
- De Grood [10]: "... valid and invalid equivalence *categories* ..."

Let's have a look at the mathematical origin of the term partition, which is "set theory". I could not find my old schoolbooks, but Wikipedia tells me that:

> "A partition P of a set S is a collection of pairwise disjoint nonempty sets such that ∪P=S."

and

> "Any partition P of a set S introduces an equivalence relation on S, where each A∈P is an equivalence class. Similarly, given an equivalence relation on S, the collection of distinct equivalence *classes* is a partition of S."

I propose to go back to basics and stick to the mathematical terms as used in set theory.

From the mathematical lingo above we learn the following (rephrased in plain English):

- *Partitioning* is the act of chopping things up.
- The way something is chopped up into a collection of bits and pieces is called a *partition*.
- The bits and pieces themselves are called *classes*.

So when I identify two possible partitions for the integers 1 to 10, it means that I can split them up in two ways, i.e. the odd numbers and the even numbers (this is the first partition) or the prime numbers and the numbers that are not primes (this is the second partition). Separating the numbers smaller than 5 from the numbers greater than or equal to 5 yields a third partition also consisting of two classes.

Valid and invalid

Once the input domain for a program has been partitioned into equivalence classes, we have to decide whether these classes are valid or invalid before we can combine them into test cases. But how do we define "valid"?

- Black [9]: "... *valid* classes ... describe valid situations that the system should handle normally ..."
- Van Veenendaal [6]: "*Invalid* data in the context of EP does not mean that the data is incorrect: it means that this data lies outside a specific partition"
- Burnstein [7]: "*Invalid* classes represent erroneous or unexpected inputs"

I have also been told that a class should be considered invalid when the behavior of the program is not specified for input from that class. But what if the specified behavior is an error message? D.J. de Grood [10] goes as far as saying: "... is not an invalid value *because the error handling for this input value has been specified* ...".

I propose to adopt and adapt the definitions as applied in the classification tree method.

"Valid *classes* describe input situations which are processed regularly by the test object. Invalid *classes should* provoke an error handling reaction by the test object." Most of this quote is original. I adapted it by adding the word "should" to cover those situations where the error handling mechanism is not (yet) in place and I put in the word *classes* where Grochtmann [4] talks about "test cases".

The recipe

Slicing and dicing

It is now clear what partitioning is (slicing and dicing) and what a class is (a slice or a dice), but not yet how the input domain is to be sliced. Van Veenendaal [6] and Kit [3] give a good overview of all kinds of input and how they typically could be partitioned, but is this a tester's job? The basis for partitioning is already present in the requirements "Children under the age of 6 get free entrance when accompanied by at least one parent". We are given the partition for age and for number of accompanying parents for free. We do have to make sure, though, that the customer, the requirements engineer, and the software developer all interpret the information in the same way.

Combining classes

Now we have clarified the terminology stuff, and I hope you can agree with the choices I made, we come to the method of combining classes into test cases. There are a lot of opinions among the referred sources on EP, so again we will have to make choices.

Let's first have a look at the standard for SW component testing – BS 7925-2 [5]. This standard offers us freedom: "Two distinct approaches can be taken when generating the test cases. In the first a test case is generated for each identified partition on a one-to-one basis, while in the second a minimal set of test cases is generated that cover all the identified partitions." Note that the word *partitions* is used, where I propose to use *classes*.

Most sources create the smallest set of test cases that cover all valid classes and subsequently follow Myers' [1] approach: "Write a test case that covers one, and only one, of the uncovered invalid equivalence classes" until all invalid classes are covered as well.

The reason for not combining multiple invalid inputs into one test case is clearly indicated by Myers and quoted by Kit and Veenendaal: "If multiple invalid ECs are tested in the same test case, some of those tests may never be executed because the first test may mask other tests or terminate execution of the test case." Yet some ([6], [9]) state that a single, completely invalid test case may be useful. Especially in web ap-

plications, input data are often entered in a form and are checked for validity before being sent to the server.

Let me present to you the following menus for combining classes:

Choice 1 – a low-fat meal

- Starter: a minimal set of test cases covering all valid classes
- Main course: a minimal set of test cases covering all invalid classes
- Dessert: sorry, no dessert

Choice 2 – a regular meal

- Starter: a minimal set of test cases covering all valid classes
- Main course: a set of test cases, each covering one single invalid class at a time, until all invalid classes are covered (served)
- Dessert: sorry, no dessert

Choice 3 – a hearty meal

- Starter: a minimal set of test cases covering all valid classes
- Main course: a set of test cases, each covering one single invalid class at a time, until all invalid classes are covered (served)
- Dessert: one single test case covering only the finest invalid classes

Choice 4 – (almost) all you can eat

- Starter: a minimal set of test cases covering all valid classes
- Main course: a set of test cases, each covering one single invalid class at a time, until all invalid classes are covered (served)
- Dessert: a minimal set of completely invalid test cases covering all invalid classes
- Extras: for the extremely hungry, we offer a well chosen composition of additional test cases covering all output classes

And how do we choose? Well, it depends on what we know about the subject under test. For low risk situations, when you are not very hungry, the smallest set of test cases might be the right choice. If we know, like in the web site example above, that the software is supposed to check each value entered in the fields of a form before starting the calculation, it is wise to add that extra completely invalid test case.

For situations with a higher risk, we want to make sure that each and every one of the invalid classes is recognized as being invalid and treated as such, so we need at least a good regular meal or maybe even "all you can eat".

But how about the extras? Is output partitioning a nutritious addition, or is it just fattening?

Again, it depends on the situation and the goal of testing. When you are testing a module that translates input into output (which happens quite often) and, subsequently, that output is used as input to another module, it may be very valuable to know whether or not all possible output is going to be accepted or rejected for the right reasons. In fact we are talking integration testing here. In order to execute a test like this, all possible input classes for the accepting module must be generated. So we need to know all about the output of the calling module. The range of the calling module must fit with the domain of the called module [2]. As Beizer puts it, "... the caller's range is the caller's notion of the called routine's domain ..." and we want to verify that that notion is correct.

Bon appétit!

I guess an upfront apology is appropriate. It was never my intention that the next time you eat a pizza quattro stagioni you no longer want to slice it, but you want to partition it into four classes and you expect every bite from a single class to taste *equivalently*.

References

[1] Myers, Glenford J. *The art of software testing*. 1979.

[2] Beizer, Boris. *Software Testing Techniques*. 1983.

[3] Kit, Edward. *Software Testing in the Real World*. 1992.

[4] Grochtmann, M. *Test Case Design Using Classification Trees*. 1994.

[5] *BS7925-2*. 1998.

[6] Van Veenendaal, E. *The testing practitioner*. 2002.

[7] Burnstein, Ilene. *Practical software testing*. 2003.

[8] Koomen, Tim e.a. *TMap Next*. 2006.

[9] Black, Rex. *Advanced Software Testing – Vol. 1: Guide to the ISTQB Advanced Certification as an Advanced Test Analyst*. 2008.

[10] De Grood, D.J. *Testgoal*. 2008.

> about the author

Piet de Roo has been engaged in software development and testing since 1996. He has been employed as a test analyst, test manager, test process manager, and test automation manager mainly in the automotive branch, focusing on embedded software in car radio and navigation systems. In various international roles he has been responsible for software verification in traditional and agile processes. Currently Piet works at Improve Quality Services as a consultant, coach, and teacher in testing, test management, and test automation.